Formalization and Correctness of the PALS Pattern for Asynchronous Real-Time Systems

José Meseguer[1] and Peter Ölveczky[2]

[1] University of Illinois at Urbana-Champaign
[2] University of Oslo

Abstract. Due to physical requirements, what in essence and at a higher level of abstraction is a *logically synchronous* real-time system has to be often realized as a *distributed, asynchronous system*. Getting asynchronous real-time systems right is a very *error prone and labor-intensive* task. The *Physically Asynchronous Logically Synchronous* (PALS) architectural pattern can greatly reduce the design and verification complexities of achieving logical synchrony in a distributed real-time system implementation. The PALS philosophy is to provide a *correct-by-construction* pattern of very wide applicability. The main goal of this work is to make the PALS *correctness property* —applying to a wide range of designs— mathematically precise. For this, we define a *formal model* of the PALS transformation, and give *formal requirements* for the allowed logically synchronous system designs, and for the operating environments in which a resulting PALS distributed design is to be deployed. Based on such a formal model and formal requirements, we also give a *mathematical proof of correctness* for PALS, and a *proof of optimality*, showing that the PALS period is shortest possible. The PALS proof of correctness can greatly facilitate the formal verification effort, because it reduces the verification of a complex asynchronous real-time system to that of its much simpler synchronous high-level design. Our formal model is developed in rewriting logic using the Real-Time Maude specification language. Since such formal specifications are executable, they can be used as a basis for correct-by-construction code generation implementations of PALS.

1 Introduction

Physical requirements often necessitate *distributed, asynchronous* implementations of what in essence and at a higher level of abstraction are *logically synchronous* real-time architectures. However, getting an asynchronous real-time systems right is a very *error prone and labor-intensive* task. This has a direct bearing on *safety-critical systems*, for which the cost of *certification* can also be quite high. The *Physically Asynchronous Logically Synchronous* (PALS) architecture developed by researchers at Rockwell-Collins and UIUC (see [5, 1, 8]) can greatly help in generating, from a synchronous design, a correct-by-construction asynchronous real-time implementation.

We believe that PALS, as a complexity-reducing formalized architectural pattern, can greatly increase system quality and can greatly reduce both the cost of design, implementation, and verification of distributed real-time systems, such as avionic systems, and the cost of certifying highly critical systems of this kind. Specifically, certification of Level A safety critical system requires full coverage of the verification of its design and the validation of its implementation. This may easily become unfeasible for a distributed design; but full design verification may be achieved by automatic model checking verification of the semantically equivalent, but much simpler, synchronous design thanks to the PALS formally verified architectural pattern.

This document focuses on PALS's mathematical model, its timeline and the optimality of its logical time period, the formal executable specification of the wrappers that make the PALS transformation possible, and a formal correctness proof which justifies why verification of the much simpler synchronous system ensures the correctness of its PALS asynchronous version. This work has been developed in close interaction with all the other members of the PALS team at UIUC and Roockwell-Collins and should be viewed as a key component of the PALS architecture; specifically, as the *formalization* of such an architecture. In this way, PALS becomes a *formalized architectural pattern*, which can be safely applied with very strong correctness guarantees to a very wide range of real-time synchronous designs. Given the importance of fully documenting PALS formal model and its correctness, we think that the topics we study here deserve being spelled out in detail; this complements other documents on PALS, and provides both an in-depth mathematical presentation of the PALS architecture and a detailed reference for PALS's formal properties and underpinnings.

Besides documenting the PALS mathematical aspects, a more practical application of this work is that the formally specified wrappers that are added to the components of a synchronous real-time system in order to obtain their PALS counterpart can support correct-by-construction *code generation* schemes that can be used to automatically generate an asynchronous system PALS implementation from a simpler synchronous system implementation.

Main Contributions. The main contributions of this work are:

1. A precise formal model in rewriting logic of the PALS transformation, expressed in Real-Time Maude, including precise assumptions about the allowable synchronous designs to which PALS can be applied and about the underlying operating conditions of the network and clock synchronization infrastructures.
2. A precise derivation of the PALS period, as well as a proof of its *optimality*, showing that it is shortest possible under the given assumptions about the network and clock synchronization infrastructures.
3. A *bisimulation theorem*, showing that the original synchronous design and the so-called stable states of the corresponding PALS asynchronous design constitute bisimilar systems.
4. A justification of the method that *reduces* the formal verification of temporal logic safety and liveness properties of an asynchronous PALS design

—typically unfeasible due to the enormous state space explosion caused by concurrency— to the model checking verification of its synchronous counterpart; and
5. A formal specification of the *PALS wrappers*, which are used to encapsulate the different synchronous state machines in order to obtain the corresponding asynchronous system; since such formally-specified wrappers are *executable*, they can support correct-by-construction *code generation* schemes to generate the code of a PALS system from that of its synchronous code.

The logical framework in which we have developed this model is *rewriting logic* [4], and in particular, the Real-Time Maude formal specification language [7], which supports formal executable specification and model checking verification of rewriting logic models of real-time systems.

1.1 Key Ideas of the PALS Formal Model

The formalization of PALS is achieved by:

1. Defining a synchronous system as an *synchronous ensemble of state machines* $\mathcal{E}$, connected together by a *wiring diagram*.
2. Giving a specification Γ of the following *performance bounds*:
   - the *clock skew* of the local asynchronous clocks for each state machine should be strictly smaller than ϵ
   - minimum and maximum duration times $0 \leq \alpha_{min} \leq \alpha_{max}$ for any machine to consume inputs, make a *transition*, and produce outputs;
   - minimum and maximum *message transmission delays* $0 \leq \mu_{min} \leq \mu_{max}$ for the network.
3. Defining PALS as a formal *model transformation*

$$(\mathcal{E}, \Gamma) \mapsto \mathcal{A}(\mathcal{E}, \Gamma)$$

   where a synchronous design $\mathcal{E}$ and its performance bounds Γ generate an asynchronous design $\mathcal{A}(\mathcal{E}, \Gamma)$.
4. Using rewriting logic and Real-Time Maude to make the formal asynchronous model $\mathcal{A}(\mathcal{E}, \Gamma)$ *executable*.
5. Viewing the $\mathcal{A}(\mathcal{E}, \Gamma)$ model as the executable specification of *correct wrappers* for the state machines of $\mathcal{E}$. Such executable wrapper specifications could then be used as a basis for *automatically generate code* that implements the PALS system $A(\mathcal{E}, \Gamma)$ based on an implementation or a design of $\mathcal{E}$.

All the results presented in this work, such as the time period calculation, the proof of its optimality, and the bisimulation theorem, are based on the above formal model of PALS; and on simple extensions of it, such a the associated Kripke structures that are used as the semantic basis of temporal logic properties.

The rest of this document is organized as follows. In Section 2 we summarize the basic ideas about rewriting logic and Real-Time Maude needed to define our formal model of PALS. In Section 3 we give a formal definition of the

synchronous models that are legal input designs for the PALS transformation, including precise formal defintions of typed machines, synchronous ensembles, and synchronous composition. In Section 4 we define in detail the assumptions about clock drift, network delays, and machine execution times that, together with the given synchronous ensemble are the inputs to the PALS transformation. We then give a precise time-line analysis of the period that must be chosen, based on these parameters, for the PALS asynchronous system to achieve logical synchrony. Based on this analysis, we then give a formal specification in Real-Time Maude (parametric on the input ensemble $\mathcal{E}$ and the performance bounds Γ) of the resulting PALS-transformed asynchronous system, and collect in Section 5 some facts about the consequences of the extra generality gained by allowing local clock functions that are only piecewise continuous instead of just continuous. In Section 6 we state and prove the main bisimulation result, connecting the states of the synchronous system with the so-called stable states of its PALS-transformed asynchronous counterpart; some key lemmas for this theorem are collected in Section 7. A detailed proof of the optimality of the PALS period used to achieve logical synchrony is Section 8, an some final conclusions are drawn in Section 9.

Acknowledgments. This work is part of a broader collaboration with Steve Miler and Darren Cofer at Rockwell-Collins and with Lui Sha, Abdulla Al-Nayeem, and Mu Sun at UIUC on the PALS architecture. The ideas presented here have been developed in close interaction with all these people, who have provided very useful comments on earlier versions of this work. We thank particularly Lui Sha and Mu Sun for their very careful and insightful comments on an earlier version that has led to substantial improvements. We gratefully acknowledge funding for this research from the Rockwell-Collins corporation. Partial support has also been provided by the National Science Foundation under Grants IIS 07-20482 and CNS 08-34709, and by the Research Council of Norway.

2 Real-Time Maude

A Real-Time Maude [7] *timed module* specifies a *real-time rewrite theory* of the form (Σ, E, IR, TR), where:

- (Σ, E) is a *membership equational logic* [3] theory with Σ a signature[3] and E a set of *confluent and terminating conditional equations*. (Σ, E) specifies the system's state space as an algebraic data type, and must contain a specification of a sort `Time` modeling the (discrete or dense) time domain.
- IR is a set of (possibly conditional) *labeled instantaneous rewrite rules* specifying the system's *instantaneous* (i.e., zero-time) local transitions, written `rl [`l`] :` t `=>` t', where l is a *label*. Such a rule specifies a *one-step transi-*

[3] i.e., Σ is a set of declarations of *sorts*, *subsorts*, and *function symbols*

tion from an instance of t to the corresponding instance of t'. The rules are applied *modulo* the equations E.[4]

– *TR* is a set of *tick (rewrite) rules*, written with syntax

`rl [`l`] : {`t`} => {`t'`} in time` τ `.`

that model time elapse. `{_}` is a built-in constructor of sort `GlobalSystem`, and τ is a term of sort `Time` that denotes the *duration* of the rewrite.

The initial state must be a ground term of sort `GlobalSystem` and must be reducible to a term of the form `{`t`}` using the equations in the specifications.

The Real-Time Maude syntax is fairly intuitive. For example, a function symbol f is declared with the syntax `op` f `:` $s_1 \ldots s_n$ `->` s, where $s_1 \ldots s_n$ are the sorts of its arguments, and s is its (value) *sort*. Equations are written with syntax `eq` t `=` t', and `ceq` t `=` t' `if` *cond* for conditional equations. The mathematical variables in such statements are declared with the keywords `var` and `vars`. We refer to [3] for more details on the syntax of Real-Time Maude.

In object-oriented Real-Time Maude modules, a *class* declaration

`class` C `|` att_1 `:` s_1`, ... ,` att_n `:` s_n `.`

declares a class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C in a given state is represented as a term `<` $O : C \mid att_1 : val_1, ..., att_n : val_n$ `>` of sort `Object`, where O, of sort `Oid`, is the object's *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n. In a concurrent object-oriented system, the state is a term of the sort `Configuration`. It has the structure of a *multiset* made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative, so that rewriting is *multiset rewriting* supported directly in Real-Time Maude.

The dynamic behavior of concurrent object systems is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rule

```
rl [l] : m(O,w)  < O : C | a1 : x, a2 : O', a3 : z >   =>
                 < O : C | a1 : x + w, a2 : O', a3 : z >  dly(m'(O'),x) .
```

defines a parametrized family of transitions (one for each substitution instance) message `m`, with parameters `O` and `w`, is read and consumed by an object `O` of class `C`. The transitions have the effect of altering the attribute `a1` of the object `O` and of sending a new message `m'(O')` *with delay* `x` (see [7]). "Irrelevant" attributes (such as `a3`, and the *right-hand side occurrence* of `a2`) need not be mentioned in a rule (or equation).

A *subclass* inherits all the attributes and rules of its superclasses.

[4] E is a union $E' \cup A$, where A is a set of equational axioms such as associativity, commutativity, and identity, so that deduction is performed *modulo* A. Operationally, a term is reduced to its E'-normal form modulo A before any rewrite rule is applied.

Formal Analysis. A Real-Time Maude specification is *executable*, and the tool offers a variety of formal analysis methods. The *rewrite* command simulates *one* fair behavior of the system *up to a certain duration*. It is written with syntax `(trew` t `in time <=` τ `.)`, where t is the initial state and τ is a term of sort `Time`. The *search* command uses a breadth-first strategy to analyze all possible behaviors of the system, by checking whether a state matching a *pattern* and satisfying a *condition* can be reached from the initial state. The command which searches for n states satisfying the *pattern* search criterion has syntax

`(utsearch [`n`]` t `=>*` *pattern* `such that` *cond* `.)`

Real-Time Maude also extends Maude's *linear temporal logic model checker* to check whether each behavior, possibly up to a certain time bound, satisfies a temporal logic formula. *State propositions*, possibly parametrized, can be predicates characterizing properties of the state and/or properties of the global time of the system. They are operators of sort `Prop`, and their semantics is defined by (possibly conditional) equations of the form `{`*statePattern*`} |=` *prop* `=` b, for b a term of sort `Bool`, which defines the state proposition *prop* to hold in all states `{`t`}` where `{`t`} |=` *prop* evaluates to `true`. A temporal logic *formula* is constructed by state propositions and temporal logic operators such as `True`, `False`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), and `U` ("until"). The time-bounded model checking command has syntax

`(mc` t `|=t` *formula* `in time <=` τ `.)`

for initial state t and temporal logic formula *formula* .

3 Formal Definition of the Synchronous Model

This section formally defines the synchronous model of computation as a collection of deterministic *typed machines* and an *environment*, and a set of connections that connect output ports of the machines and the environment to input ports.

3.1 Typed Machines

A *typed machine* is a component in the synchronous model.

Definition 1. *A* (deterministic) typed machine *is a 4-tuple*

$$M = (D_i, S, D_o, \delta_M)$$

where

- D_i*, called the* input set*, is a nonempty set of the form* $D_i = D_{i_1} \times \cdots \times D_{i_n}$*, for* $n \geq 1$*, where* $D_{i_1}, \ldots, D_{i_n}$ *are called the* input data types*.*
- S *is a set, called the* set of states*.*

– D_o, *called the* output set, *is a nonempty set of the form* $D_o = D_{o_1} \times \cdots \times D_{o_m}$, *for* $m \geq 1$, *where* $D_{o_1}, \ldots, D_{o_m}$ *are called the* output data types.
– δ_M, *called the* input-output-transition (i-o-t) function, *is a function*

$$\delta_M : D_i \times S \to S \times D_o$$

We call M finite *iff* D_i, S, *and* D_o *are all finite sets.*

That is, such a machine has n input ports and m output ports; an input to port k should be an element of the set D_{i_k}, and an output from port j should be an element of the set D_{o_j}. Pictorially, we represent a typed machine as a box with typed input and output wires as shown in Fig. 1.

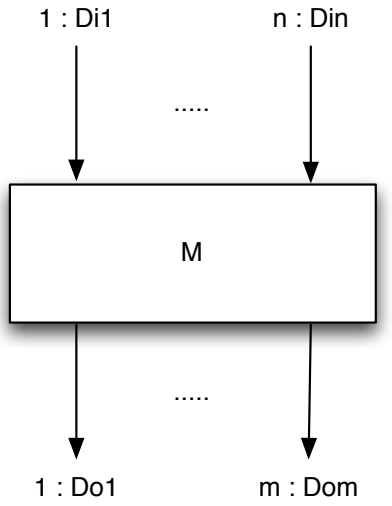

Fig. 1. Graphical representation of a machine.

3.2 Synchronous Ensembles of Typed Machines

Typed machines can be "wired together" into arbitrary sequential and parallel compositions by means of a "wiring diagram," as the one shown in Fig. 2, where the types are left implicit, but where it is assumed that the type in an output wire must match any types in the input wires connected with it:

Definition 2. *A (typed)* synchronous machine ensemble *is a 4-tuple*

$$\mathcal{E} = (J \cup \{e\}, \{M_j\}_{j \in J}, E, src)$$

where

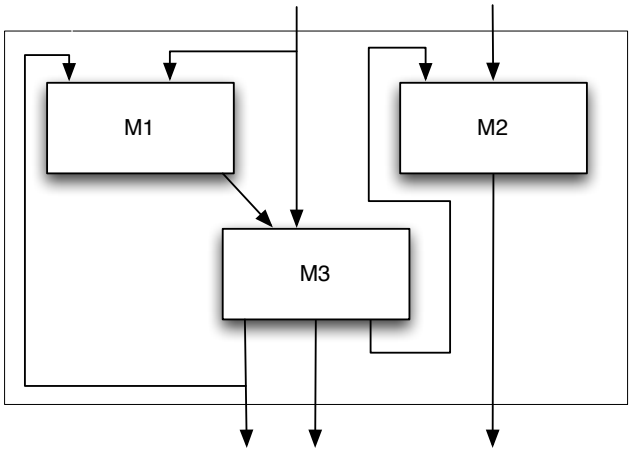

Fig. 2. A machine ensemble.

- *J is a finite set, called the set of* indices*, and e is an element, called the* environment index*, with $e \notin J$.*
- *$\{M_j\}_{j \in J}$ is a J-indexed family of typed machines.*
- *E, called a* typed environment*, is an ordered pair of sets*

$$E = (D_i^e, D_o^e)$$

where D_i^e, called the environment's input set *(inputs* to *the environment), is a nonempty set of the form*

$$D_i^e = D_{i_1}^e \times \cdots \times D_{i_{n_e}}^e, \text{ for } n_e \geq 1$$

and D_o^e, called the environment's output set *(outputs* from *the environment), is a nonempty set of the form*

$$D_o^e = D_{o_1}^e \times \cdots \times D_{o_{m_e}}^e, \text{ for } m_e \geq 1$$

- *src is a function that assigns to each input port (j, n) (the input port number n of machine j) the corresponding output port (or "source" for that input) $src(j, n)$. Formally, we define the set of input ports and output ports, respectively, as follows:*
  - $In_{\mathcal{E}} = \{(j, n) \in (J \cup \{e\}) \times \mathbb{N} \mid 1 \leq n \leq n_j\}$
  - $Out_{\mathcal{E}} = \{(j, n) \in (J \cup \{e\}) \times \mathbb{N} \mid 1 \leq n \leq m_j\}$

  Then src is a surjective function

$$src : In_{\mathcal{E}} \to Out_{\mathcal{E}}$$

assigning to each input port the output port to which it is connected, and such that "the types match". That is, if we denote by $D_{i_k}^j$ the set of data allowed

as input in the kth input port of machine M_j (resp. kth input port of the environment if $j = e$), and same with output ports, then if $src(j, q) = (k, l)$ we should have $D^k_{o_l} \subseteq D^j_{i_q}$.
In addition, we require that there are no self-loops from the environment to itself; that is, for $(e, q) \in In_{\mathcal{E}}$, if $src(e, q) = (k, p)$, then $k \in J$.

As its name suggests, a synchronous ensemble $\mathcal{E}$ has a *lock-step synchronous semantics*, in the sense that the state-and-output transitions of all the machines are performed simultaneouly, and whenever a machine has a feedback wire to itself and/or to any other machine, then the corresponding output becomes an input for any such machine at the *next* instant. As explained below, what this means mathematically is that any ensemble $\mathcal{E}$ is semantically equivalent to a *single state machine*, called the *synchronous composition* of all the machines in the ensemble $\mathcal{E}$. This has enormous practical importance for formal verification purposes, since the composed state machine is much simpler than an asynchronous system realizing such a design in a distributed way. In particular, model checking a single state machine is much more efficient and feasible than verifying a system of asynchronously interacting machines, which can esily become unfeasible due to the combinatorial explosion caused by the system's concurrency.

Example 1. In the machine ensemble in Fig. 2,

- $J = \{1, 2, 3\}$
- $\{M_j\}_J$ is the mapping $1 \mapsto M_1,\ 2 \mapsto M_2,\ 3 \mapsto M_3$.
- $n_e = 3$ (the number of inputs to the environment) and $m_e = 2$ (number of outputs from the environment).
- With an ordering of ports from left to right, the wiring function *src* is:
  - $(1, 1) \mapsto (3, 1)$
  - $(1, 2) \mapsto (e, 1)$
  - $(2, 1) \mapsto (3, 3)$
  - $(2, 2) \mapsto (e, 2)$
  - $(3, 1) \mapsto (1, 1)$
  - $(3, 2) \mapsto (e, 1)$
  - $(e, 1) \mapsto (3, 1)$
  - $(e, 2) \mapsto (3, 2)$
  - $(e, 3) \mapsto (2, 1)$

Intuitively, we can enclose the typed machines M_1, M_2, and M_3 in the box with thin lines, hiding the internal details of how the machine ensemble is decomposed. The single machine resulting in this way from the composition of machines M_1, M_2, and M_3 is called the *synchronous composition* of M_1, M_2, and M_3, according to the given wiring diagram, and can itself be seen as a typed machine. The general definition is as follows.

Definition 3. *Given a synchronous machine ensemble $\mathcal{E} = (J \cup \{e\}, \{M_j\}_{j \in J}, E, src)$, its* synchronous composition *is the typed machine*

$$M_{\mathcal{E}} = (D^{\mathcal{E}}_i, S^{\mathcal{E}}, D^{\mathcal{E}}_o, \delta_{\mathcal{E}})$$

where

- $D_i^{\mathcal{E}} = D_o^e$ *(the input set of the composed machine is the output set of the environment)*
- $D_o^{\mathcal{E}} = D_i^e$ *(the output set is the input set of the environment)*
- $S^{\mathcal{E}} = (\Pi_{j\in J} S_j) \times (\Pi_{j\in J} D_{OF}^j)$*, where if* $D_o^j = D_{o_1}^j \times \cdots \times D_{o_{m_j}}^j$ *is the output set of* M_j*, then* D_{OF}^j *is the set* $D_{OF}^j = D_{OF_1}^j \times \cdots \times D_{OF_{m_j}}^j$*, where, for* $1 \leq m \leq m_j$*,* $D_{OF_m}^j = D_{o_m}^j$ *if* $(j,m) = src(l,q)$ *for some* $l \in J$*, and* $D_{OF_m}^j = 1$ *otherwise, with* $1 = \{*\}$ *a one point set. Intuitively,* D_{OF}^j *stores the "feedback outputs" of machine* M_j*. We then have an obvious "*feedback output*" function*

$$fout_j : D_o^j \to D_{OF}^j$$

where for $1 \leq m \leq m_j$*, we have* $\pi_m(fout_j(d_1, \ldots, d_{m_j})) = d_m$ *if* $(j,m) = src(l,q)$ *for some* $l \in J$*, and* $\pi_m(fout_j(d_1, \ldots, d_{m_j})) = *$ *otherwise, with* π_m *the* m*-th projection from the Cartesian product* D_{OF}^j*. Similarly, for each* $k \in J$ *we have an obvious* input *function*

$$in_k : D_o^e \times \Pi_{j\in J} D_{OF}^j \to D_i^k$$

where for $1 \leq n \leq n_k$*, with* $src(k,n) = (l,q)$*, we have* $\pi_n(in_k(\boldsymbol{d}, \{\boldsymbol{d_j}\}_{j\in J})) =$ *if* $l = e$ *then* $\pi_q(\boldsymbol{d})$ *else* $\pi_q(\boldsymbol{d_l})$ *fi, where* π_q *denotes the* q*-th projection from the corresponding Cartesian product.*
- *The i-o-t function for* $M_{\mathcal{E}}$ *is the function*

$$\delta_{\mathcal{E}} : D_i^{\mathcal{E}} \times S^{\mathcal{E}} \ \to \ S^{\mathcal{E}} \times D_o^{\mathcal{E}}$$

where for each $(\boldsymbol{d}, (\{s_j\}_{j\in J}, \{\boldsymbol{d_j}\}_{j\in J})) \in D_i^{\mathcal{E}} \times S^{\mathcal{E}}$ *we define* $\delta_{\mathcal{E}}(\boldsymbol{d}, (\{s_j\}_{j\in J}, \{\boldsymbol{d_j}\}_{j\in J})) = ((\{s'_j\}_{j\in J}, \{\boldsymbol{d'_j}\}), \boldsymbol{d'})$*, where for each* $l \in J$ *we have*

$$\begin{array}{l} s'_l = \pi_1(\delta_{M_l}(in_l(\boldsymbol{d}, \{\boldsymbol{d_j}\}_{j\in J}), s_l)) \\ \boldsymbol{d'_l} = fout_l(\pi_2(\delta_{M_l}(in_l(\boldsymbol{d}, \{\boldsymbol{d_j}\}_{j\in J}), s_l))) \end{array}$$

and for each $1 \leq n \leq n_e$ *with* $src(e,n) = (j', r)$

$$\pi_n(\boldsymbol{d'}) = \pi_r(\pi_2(\delta_{M_{j'}}(in_{j'}(\boldsymbol{d}, \{\boldsymbol{d_j}\}_{j\in J}), s_{j'}))).$$

3.3 Environment Constraints.

In our model of the behaviors of a system, we assume a nondeterministic environment where there could be some constraints on the values generated by this environment. In this report, we assume that the environment constraint can be defined as a predicate

$$c_e : D_o^e \to Bool$$

so that $c_e(d_1^e, \ldots, d_{o_{m_e}}^e)$ is $true$ if and only if the environment can generate output $(d_1^e, \ldots, d_{o_{m_e}}^e)$.

3.4 The Transition System and the Kripke Structure Associated to a Machine Ensemble

To each machine ensemble that operates in an environment with a given environment constraint, we can associate a transition system defining the behaviors of the system.

Definition 4. *Given a machine ensemble* $\mathcal{E} = (J \cup \{e\}, \{M_j\}_{j \in J}, E, src)$ *with environment constraint* c_e*, the corresponding transition system is defined as a pair* $(S^{\mathcal{E}} \times D_i^{\mathcal{E}}, \longrightarrow_{\mathcal{E}_{c_e}})$*, where the transition relation* $\longrightarrow_{\mathcal{E}_{c_e}}$ *is defined by*

$$(\boldsymbol{s}, \boldsymbol{i}) \longrightarrow_{\mathcal{E}_{c_e}} (\boldsymbol{s'}, \boldsymbol{i'})$$

iff a machine ensemble in state $\boldsymbol{s}$ *and with input* $\boldsymbol{i}$ *from the environment has a transition to state* $\boldsymbol{s'}$*, and the environment can generate output* $\boldsymbol{i'}$ *in the next step:*

$$(\boldsymbol{s}, \boldsymbol{i}) \longrightarrow_{\mathcal{E}_{c_e}} (\boldsymbol{s'}, \boldsymbol{i'}) \iff \boldsymbol{s'} = \pi_1(\delta_{\mathcal{E}}(\boldsymbol{i}, \boldsymbol{s})) \wedge c_e(\boldsymbol{i'}).$$

Let $\mathcal{E}$ be a typed machine ensemble with environment constraint c_e, AP a set of *atomic propositions*, and let $L : S^{\mathcal{E}} \times D_i^{\mathcal{E}} \to \mathcal{P}(AP)$ be a labeling function that assigns to each state $(s, i) \in S^{\mathcal{E}} \times D_i^{\mathcal{E}}$ the set $L(s, i)$ of atomic propositions that hold in (s, i). Then $(S^{\mathcal{E}} \times D_i^{\mathcal{E}}, \longrightarrow_{\mathcal{E}_{c_e}}, L)$ is the Kripke structure associated to $(\mathcal{E}, c_e, L)$.

Notice that, since the machine ensemble's output is a function of the machine state (including content in the feedback loops) and the input from the environment, we may also define atomic propositions that talk about the output from the machines. For example, we can define a *parametric* atomic proposition *out*, so that $out(v)$ holds in a state $(\boldsymbol{s}, \boldsymbol{i})$ if the machine ensemble generates output $\boldsymbol{v}$ to the environment, can be defined by $out(\boldsymbol{v}) \in L(\boldsymbol{s}, \boldsymbol{i}) \iff \boldsymbol{v} = \pi_2(\delta_{\mathcal{E}}(\boldsymbol{i}, \boldsymbol{s}))$.

4 The PALS Asynchronous Model

This section presents the rewriting logic model of the asynchronous PALS transformation of a synchronous machine ensemble in detail.

Section 4.1 makes explicit some assumptions about clock drifts and computation and communication times, and defines some constant values. Section 4.2 gives a brief overview of the asynchronous system, and Section 4.3 focuses on the time line. Sections 4.4 to 4.8 then specify the asynchronous model as a rewrite theory in Real-Time Maude. Finally, Section 4.9 defines the initial states.

4.1 Some System Assumptions

Time Domain. The type of time (discrete or dense) is a parameter of the model. It could be $\mathbb{N}$, or $\mathbb{R}_{\geq 0}$, or $\mathbb{Q}_{\geq 0}$, for example. (If it is $\mathbb{N}$, we can scale up things so that one "logical time step" can correspond to many basic steps.) For simplicity and fullest generality, in what follows we will assume that all is done in $\mathbb{R}_{\geq 0}$.

Clock Drifts and Clock Synchronization. A basic assumption is that a clock synchronization algorithm is executing "in the background" and guarantees a certain bound on the imprecision of the local clocks. We assume an *external clock synchronization*; that is, the difference between the time of a local clock and "real" global time is always *strictly less* than a given bound ϵ.

To reason about clock drift in a general way, we assume a local clock function c_j for each machine M_j that assigns to each global instant r the local clock value $c_j(r)$:

Definition 5. *A function $c : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ is called an* ϵ-drift local clock function *if and only if the following conditions are satisfied:*

1. *c is monotonic and piecewise continuous,*
2. *$\forall x \in \mathbb{R}_{\geq 0},\ |c(x) - x| < \epsilon$, and*
3. *$\forall x \in \mathbb{R}_{\geq 0},\ \inf\{t \mid c(t) \geq x\} \in \{t \mid c(t) \geq x\}$.*

The assumption of monotonicity is very good to have, particularly to have a clear idea of when and where to tick the global clock. Synchronization can be achieved while preserving monotonicity. The idea is the same as when one has an expensive clock that typically does not allow to be wound *backwards*. How do you adjust such a clock to the precise time? Well, if the clock is too fast, then you can just "stop" your clock and wait until "real" time has caught up with your clock time. If the clock is slow, then you quickly adjust it forward to the "real" time. This is an isolated discontinuity which happens only from time to time. Furthermore, the third condition above ensures that the time at which a discontinuity of a local clock function c occurs is well-defined. Alternatively, one can achieve effect of the ϵ-drift local clock function c to always be *continuous* by increasing the "ticking rate" of the local clock by a small factor whenever the clock is detected to be slow, and likewise decreasing its "ticking rate" by a small factor whenever it is detected to be fast. Continuity is preferable for applications where control of physical parameters is involved; but our results, although having a slightly simpler formulation for the continous case, do not require continuity, but only piecewise continuity of the local clocks satisfying (1)–(3) above.

Execution Times. The shortest, respectively longest, time required for *processing input, executing a transition, and generating output* is supposed to be, respectively, α_{min} and α_{max}.

Network Delays. The point-to-point message transmission time is assumed to always be greater than or equal to a minimum value $\mu_{min} \geq 0$, and smaller than or equal to some maximum time value $\mu_{max} \geq \mu_{min}$.

4.2 The Asynchronous System with Clock Drifts: Overview

The system is made out of a J-indexed family of objects and an environment, with each object behaving like an "asynchronous typed machine" whose inputs

and outputs are received and sent by *asynchronous message passing*. The system is supposed to execute in rounds according to "ticks" of a "logical clock." Let T denote the time between such "ticks of the logical clock" (or the *period* of the logical clock). We often write t_i for the time of the ith tick of the logical clock; i.e., $t_i = i \cdot T$.

Each object j is equipped with two timers:

`roundTimer` is a timer that should expire at the tick of each logical clock; that is, at the end of each period.

`outputBackoffTimer` is a backoff timer used to ensure that output from a machine is not sent into the network too early.

The actions of each object j can be summarized as follows:

- When `roundTimer` expires, input from the input buffer is read, a transition is executed, and the generated output is put in the output buffer. In addition, this timer is reset to the value T (denoting the period of the "logical clock").
- When the `outputBackoffTimer` timer expires, the messages in the output buffer are sent, *provided that they have been generated*. If the execution of the transition generating the outgoing messages is not yet finished when the timer expires, then the messages are sent when the execution of the transition is finished. This timer is started and set to $dly_{out} = 2 \cdot \epsilon$ `monus` μ_{min}, where `monus` is defined by x `monus` $y = \max(0, x - y)$, each time the `roundTimer` expires.

4.3 Time-line Analysis

The "time-line analysis" for object j is therefore as follows:

1. At each *local* logical clock tick (that is, when the local clock c_j shows t_i), the object gets the messages from the input buffer, executes the transition, and puts the output messages in the output buffer. This starts somewhere in the global time interval $(t_i - \epsilon, t_i + \epsilon]$ for round i. This process may end at any global time in the interval $(t_i - \epsilon + \alpha_{min}, t_i + \epsilon + \alpha_{max}]$.
2. Since the messages cannot be sent into network before the backoff timer expires, and before the messages are "ready," the messages from the output buffers are therefore sent into the network at a global time that is strictly greater than $\max(t_i + \epsilon - \mu_{min},\ (t_i - \epsilon) + \alpha_{min})$ and is less than or equal to $\max(t_i + 3 \cdot \epsilon - \mu_{min},\ (t_i + \epsilon) + \alpha_{max})$.
3. At any global time in the time interval $(\max(t_i + \epsilon,\ (t_i - \epsilon) + \alpha_{min} + \mu_{min}),\ \max(t_i + 3 \cdot \epsilon - \mu_{min},\ (t_i + \epsilon) + \alpha_{max}) + \mu_{max}]$, a message could arrive at an object, at which time it is entered into the object's input buffer.
4. When the local clock shows t_{i+1}, the object starts all over from the first point above.

An overview of this timeline is depicted in Fig. 3. It is worth remarking that some of the time intervals are right-closed. For example, the "local clock tick" in

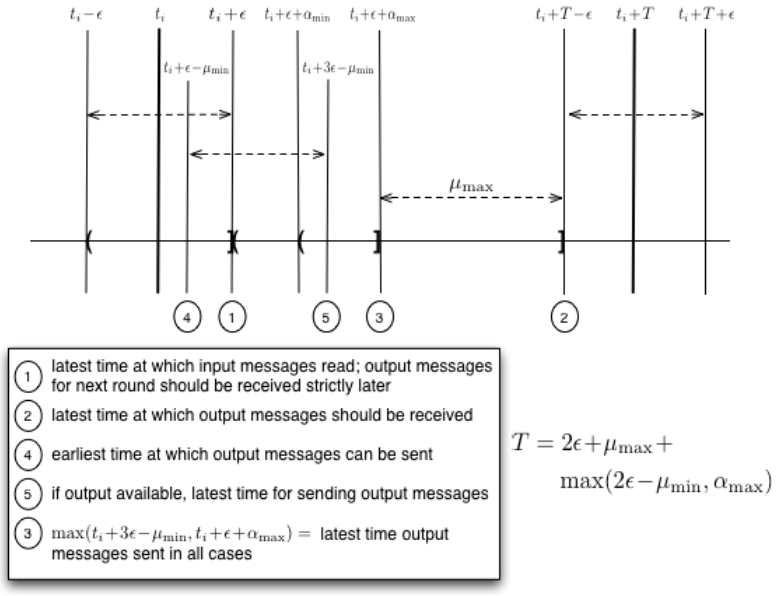

Fig. 3. PALS timeline.

item (1) above could happen at any time in the global time interval $(t_i-\epsilon, t_i+\epsilon]$ instead of the right-open interval $(t_i-\epsilon, t_i+\epsilon)$ that might seem more intuitive, given that the clock synchronization should ensure a difference that is *strictly smaller* than ϵ between the local clock time and global time. However, the "local tick" could happen at time $t_i+\epsilon$ as well: At all global times $t_i+\epsilon-\Delta$ for small $\Delta > 0$, the local clock shows $t_i - \Delta/2$, and at global time $t_i+\epsilon$ the local clock jumps to, say $t_i+\epsilon$. This scenario satisfies the assumptions on the clock synchronization, yet the "local logical tick" happens at time $t_i+\epsilon$. Section 5 gives further explanations on this issue.

Constraints. For this to work, we must ensure that messages generated for round $i+1$ should be received sometime in the global time interval $(t_i+\epsilon, t_{i+1}-\epsilon]$, which is $(t_i+\epsilon, t_i+T-\epsilon]$. Therefore, the message arrival interval $(\max(t_i+\epsilon,\ (t_i-\epsilon)+\alpha_{min}+\mu_{min}),\ \max(t_i+3\cdot\epsilon-\mu_{min},\ (t_i+\epsilon)+\alpha_{max})+\mu_{max}]$ must be a subset of $(t_i+\epsilon, t_i+T-\epsilon]$. This implies that we must have

1. $t_i + \epsilon \leq (\max(t_i + \epsilon,\ (t_i - \epsilon) + \alpha_{min} + \mu_{min})$, and
2. $\max(t_i + 3 \cdot \epsilon - \mu_{min},\ (t_i + \epsilon) + \alpha_{max}) + \mu_{max} \leq t_i + T - \epsilon$.

(1) holds trivially. (2) implies that

$$T \geq \mu_{max} + 2 \cdot \epsilon + \max(2 \cdot \epsilon - \mu_{min},\ \alpha_{max}).$$

4.4 Some Sorts

From here on, we present the formal specification of the asynchronous PALS system associated to a synchronous machine ensemble with environment constraint c_e, under the assumptions in Section 4.1, as a rewrite theory in Real-Time Maude. We start by discussing some sorts used in this specification.

Local States. The state component of a machine j has sort S_j. For convenience, we add a supersort `State` of all such states:

```
sort State .     --- supersort of local states
subsorts S_1 ... S_|J| < State .
```

It may take some time to compute the next local state of a machine. During this transition computation time, the local state has the value `[`s`,` t`]`, where s is the next state, and t is the time remaining until the execution of the transition is finished. Such a term `[`s`,` t`]` is called a *delayed state*, where the sort `DlyState` is defined as follows:

```
sort DlyState .
subsort State < DlyState .

op [_,_] : State Time ->  DlyState [ctor right id: 0] .
```

Likewise, during the execution of a transition generating the messages for the next round, these messages are not yet ready to be sent, and hence the output buffer has the value `[`$msgs$`,` t`]`, which we call a *delayed configuration*:

```
sort DlyConfiguration .
subsort Configuration < DlyConfiguration .

op [_,_] : Configuration Time ->  DlyConfiguration [ctor] .
```

The sort `Configuration` denotes multisets of objects and messages, formed with an "empty syntax" multiset union operator "`__`" (juxtaposition) which is declared to be *associative* (`assoc`) and *commutative* (`comm`) and have identity `none` (`id: none`).

We also introduce a supersort `Data` of the sorts $D_1 \ldots D_n$ of the data in the wires:

```
sort Data .
subsorts D1 ... Dn < Data .
```

Each object is also assumed to know its local wiring diagram; that is, which objects and ports are connected to its output ports in the synchronous system. This knowledge is stored in a data structure called a *local wiring*, where the sort `LocalWiring` is defined as follows:

```
sort LocalWiring .

op _-->_._ : Nat Oid Nat -> LocalWiring [ctor] .
op noWiring : -> LocalWiring [ctor] .
op _;_ : LocalWiring LocalWiring -> LocalWiring
                                    [ctor assoc comm id: noWiring] .
```

Here, a connection p `-->` j `.` p' says that the output port p of the current object is connected to the input port p' of object j. A *local wiring* is then a set of such single connections formed with the associative-commutative union operator `_;_` with identity the empty set constant `noWiring`.

4.5 The Class Declarations

Each machine M_j is translated into an object instance of a subclass $C_{[j]}$ of the class `Machine` declared as follows:

```
class Machine | state : DlyState,
                inBuffer : MsgConfiguration,
                outBuffer : DlyConfiguration,
                roundTimer : Time,
                outputBackoffTimer : TimeInf,
                clock : Time,
                localWiring : LocalWiring .

class C1 .
...
class Ck .

subclass C1 ... Ck < Machine .
```

Note that several typed machines, say, $M_{j_1}, \ldots, M_{j_r}$, can all be of the same type, and can therefore all belong to the same subclass, i.e., $C_{[j_1]} = \cdots = C_{[j_r]}$. The `state` attribute denotes the local state of the machine. The `inBuffer` attribute is the buffer of incoming messages. `outBuffer` is the output message buffer. The timers `roundTimer` and `outputBackoffTimer` have been explained above. The `clock` attribute shows the value of the local clock of the object. Finally, the `localWiring` attribute assigns to each output port number the set of input ports to which this port is connected. However, notice that here a connection

is only a *reference* for asynchronous message passing, and not a real "wired" connection as in the synchronous model.

We assume that the environment also has input and output buffers, and that it satisfies the same timing requirement as all the other objects. The environment is therefore modeled as an object instance of a class `Env` that is declared as a subclass of `Machine`:

```
class Env .

subclass Env < Machine .
```

Since we do not explicitly represent the internal "state" of the environment, the `state` attribute for the environment is given the constant default value `*`:

```
op * : -> State [ctor] .
```

4.6 Messages

Messages have the general form

`to` j `from` j' `(`p`,` d`)`

where $j, j' \in J \cup \{e\}, 1 \leq p \leq n_j$, and $d \in D^j_{i_p}$. Therefore, p is the input port of the intended recipient j, where data d from j' is to be received.

We also use the `dly` operator on messages to model the delay of such messages when they are in transit through the network, as explained in [7].

4.7 The Instantaneous Rewrite Rules

The following actions of the system are modeled by corresponding *instantaneous* rewrite rules:

1. Receive an incoming message and put it into the `inBuffer`.
2. When the `roundTimer` expires, the `inBuffer` is emptied, a transition is applied, and the output is put into the `outBuffer`. Note that, since performing a transition takes time, as already mentioned this is manifested by the fact that the results are "delayed".
3. When the `outputBackoffTimer` expires if the generated output is ready to be sent, then the contents in the output buffer are sent into the network, with appropriate message delays.
4. Otherwise, as soon as the generated output is ready to be sent *after* the `outputBackoffTimer` has expired (i.e., the `outputBackoffTimer` has the infinity value `INF`, as in the rule `outputMsg2` below), then the generated output is sent into the network.

Receive a Message. A message is received by an object and is inserted into its `inBuffer`:

```
vars j j' : Oid .
var B : MsgConfiguration .   --- multiset of messages
var p : Nat .
var d : Data .
var S : State .

rl [receiveMsg] :
   (to j from j' (p, d))
   < j : Machine | inBuffer : B, state : S >
  =>
   < j : Machine | inBuffer : B (to j from j' (p, d)) > .
```

Given that `S` has sort `State`, this rule can only be applied when a transition is not executing, i.e., when the state is not "delayed".

Reading Input and Executing a Transition. When `roundTimer` expires, the messages `B` in the `inBuffer` are read, and a transition is taken. Since different classes will have different transitions, executing transitions is modeled by a *family* of rewrite rules, one for each class $C_{[j]}$. Notice that the resulting state and messages are delayed by a value $\alpha_{min} \leq$ `X-DLY` $\leq \alpha_{max}$. In addition, the `roundTimer` must be reset to expire at the same time in the next round; i.e., it must be reset to the round time T, adjusted for possible clock jumps as explained in Section 5. Likewise, the `outputBackoffTimer` must be set to $2 \cdot \epsilon - \mu_{min}$:

```
vars X-DLY LT : Time .
var S : Data .
var W : LocalWiring .

crl [applyTrans] :
    < j : C_[j] | inBuffer : B, roundTimer : 0, state : S,
                  localWiring : W, clock : LT >
   =>
    < j : C_[j] | inBuffer : none,
                  state : [π1(δ_Mj(vect_[j](B), S)), X-DLY],
                  roundTimer : T - adjust(LT, T, 0),
                  outputBackoffTimer :
                    (2 · ε monus μ_min) monus adjust(LT, T, 0),
                  outBuffer : [makeMsg(j, W, π2(δ_Mj(vect_[j](B), S))), X-DLY] >
   if X-DLY >= α_min and X-DLY <= α_max .
```

Here, given a complete set `B` of messages of the form

$$\texttt{(to } j \texttt{ from } j'_1(1, d_1)\texttt{)} \quad \ldots \quad \texttt{(to } j \texttt{ from } j'_{n_j}(n_j, d_{n_j})\texttt{)} \quad (\dagger)$$

the function $vect_{[j]}$(B) maps B to the vector of inputs $(d_1, \ldots, d_{n_j})$. makeMsg is the obvious (but a bit tedious to spell out in detail) function that looks at the local wiring diagram W, takes the vector of output data from j, and produces the set of messages for the machines and environment getting inputs from that wire. For example, for the system in Fig. 2, makeMsg$(3, src, (d1, d2, d3))$ produces the message configuration

```
(to 1 from 3 (1, d1))
(to e from 3 (1, d1))
(to e from 3 (2, d2))
(to 2 from 3 (3, d3))
```

Expiration of the outputBackoffTimer**.** When the outputBackoffTimer expires, and the messages are already generated (that is, the output buffer matches MSG MSGS), the messages in the output buffer are sent into the network one by one, each message with its own nondeterministic delay (NTW-DLY). The timer is turned off (i.e., set to the infinity value INF) after the last message has been sent:

```
var MSGS : Configuration .
var MSG : Msg .
var NTW-DLY : Time .

crl [outputMsg1] :
    < j : Machine | outBuffer : MSG MSGS, outputBackoffTimer : 0 >
   =>
    < j : Machine | outBuffer : MSGS,
                    outputBackoffTimer :
                      if MSGS == none then INF else 0 fi >
    dly(MSG, NTW-DLY)
   if μmin <= NTW-DLY and NTW-DLY <= μmax .
```

If the execution of the transition and the generation of the outgoing messages is not finished when the outputBackoffTimer expires (that is, the output buffer has the form $[msgs, t]$ for $t > 0$), the timer is just turned off:

```
var NZT : NzTime .

rl [turnOffOutTimer] :
   < j : Machine | outBuffer : [MSGS, NZT], outputBackoffTimer : 0 >
  =>
   < j : Machine | outputBackoffTimer : INF > .
```

End of a Transition with Possible Immediate Output. When the execution of a transition and the generation of outgoing messages is finished, the "delay" of the generated messages in the output buffer is 0. If, in addition,

the `outputBackoffTimer` has expired (is `INF`), then the messages in the output buffer are immediately sent into the network one by one, each message with its own delay (rule `outPutMsg2`); otherwise the outgoing messages are kept in the output buffer, but the delay wrappers on the generated messages disappear (rule `transitionFinished`):

```
crl [outputMsg2] :
    < j : Machine | outBuffer : [MSG MSGS, 0],
                    outputBackoffTimer : INF >
   =>
    < j : Machine | outBuffer :
                      if MSGS == none then none else [MSGS, 0] fi >
    dly(MSG, NTW-DLY)
   if μmin <= NTW-DLY and NTW-DLY <= μmax .

var TI : TimeInf .

rl [transitionFinished] :
   < j : Machine | outBuffer : [MSGS, 0], outputBackoffTimer : T >
  =>
   < j : Machine | outBuffer : MSGS > .
```

Environment Behavior. Since the environment class `Env` is a subclass of `Machine`, the environment inherits the rules for receiving and sending messages. The "machine" rules for reading the input buffer and executing a transition are replaced by one rule that consumes the messages in the input buffer, and generates the output nondeterministically, but ensuring that the environment constraint c_e is satisfied:

```
var D1 : D^e_o1 .
...
var DME : D^e_o_me .

crl [consumeInputAndGenerateOutput] :
    < e : Env | inBuffer : B, roundTimer : 0, clock : LT, wiring : W >
   =>
    < e : Env | inBuffer : none,
                roundTimer : T - adjust(LT, T, 0),
                outputBackoffTimer :
                   (2 · ε monus μmin) monus adjust(LT, T, 0),
                outBuffer : [makeMsg(e,W,(D1, ..., DME)), X-DLY] >
  if ce(D1, ..., DME) /\ X-DLY >= αmin and X-DLY <= αmax .
```

4.8 Time Behavior

This section describes the time behavior of the asynchronous system.

State and Tick Rule. The global state of the system has the form {C; t}, where C is the configuration consisting of the objects and messages in the asynchronous system, and t is the global time.

The tick rule, advancing the global time in the system, is the following slight modification of the "usual" tick rule for object-oriented systems [7]:

```
var C : Configuration .
vars T T' : Time .

crl [tick] :
    {C ; T} => {delta(C, T, T') ; T + T'}  in time T'
       if T' <= mte(C, T) .
```

Here, `delta` is the function that defines how the *passage of time affects the state*, and `mte` is the function that defines the *smallest time until a timer becomes zero*. These functions are declared in the expected way:

```
op delta : Configuration Time Time -> Configuration [frozen (1)] .
op mte : Configuration Time -> TimeInf [frozen (1)] .
```

These functions distribute over the objects and messages in the state in the expected way, and must be defined for individual objects and messages.

We first define `delta`: how does time elapse affect the timers? If from time t, time advances by Δ, then the local clock has advanced from $c_j(t)$ to $c_j(t+\Delta)$, that is, by $c_j(t+\Delta) - c_j(t)$, where $c_j(t)$ is assumed to be the value `T4` given in the `clock` attribute:

```
vars t Δ T1 T2 T3 T4 T5 : Time .

eq delta(< j : Machine | roundTimer : T1, outputBackoffTimer : TI,
                         state : [S, T3], clock : T4,
                         outBuffer : [MSGS, T5] >, t, Δ) =
   < j : Machine | roundTimer : T1 monus (c_j(t + Δ) - T4),
                   outputBackoffTimer : TI monus (c_j(t + Δ) - T4),
                   state : [S, T3 monus Δ],
                   clock : c_j(t + Δ),
                   outBuffer : [MSGS, T5 monus Δ] > .

eq delta(< j : Machine | roundTimer : T1, clock : T4,
                         outBuffer : none >, t, Δ) =
   < j : Machine | roundTimer : T1 monus (c_j(t + Δ) - T4),
                   clock : c_j(t + Δ) > .
```

As for `mte`, it will be smallest of the `mte`'s of objects and messages in the configuration, where the `mte` of an object is just defined as the smallest time until one of the two timers become zero, or until the delay on the outgoing messages in the output buffer reaches 0. This is not a constructive definition, but this

just reflects the fact that we do not model the underlying clock synchronization "constructively". The assumption of monotonicity of the local clock functions is crucial to make this definition of `mte` well defined.

Finally, defining `delta` and `mte` on messages is trivial:

```
eq delta(dly(MSG, T1), T2, T) = dly(MSG, T1 monus T) .
eq mte(dly(MSG, T1), T2) = T1 .
```

4.9 Initial States

We define the the initial states of the system to start at time t_0, defined by $t_0 = T - \epsilon$. We do not start at time 0 since:

- local clocks could be less than the global clock;
- transitions are only taken (and hence output produced) when input is available.

At global times $(i \cdot T) + t_0$, for all $i \in \mathbb{N}$, the state components are undelayed and consistent, and all the input buffers are full.

What are the "initial" values of the object attributes at time t_0?

- The `clock` attribute of each object is $c_j(T - \epsilon)$, and we have $0 \leq T - 2\epsilon < c_j(T - \epsilon) < T$.
- The `outputBackoffTimer` should be turned off at this time, as all outgoing messages have been sent.
- The `roundTimer`, which is supposed to expire at each (local) time $i \cdot T$ should be initialized to $T - c_j(t_0)$, where it follows from the above clock values in the initial state that $0 < T - c_j(t_0) < 2\epsilon$.
- The `state` attribute should be an initial state of the expected sort.
- The `inBuffer`s are full of messages.
- The `outBuffer`s should be empty.
- There are no messages in transit in the network.
- The local wiring is constant and maps the ports to the input wires as illustrated below.
- The initial input $(d^e_{o_1}, \ldots, d^e_{o_{m_e}})$ from the environment should satisfy the environment constraint c_e.

In addition, the input buffers should be consistent w.r.t. the *src* function. That is, if $src(j, l) = src(j', l') = (j'', l'')$, then the data value d in the message `(to` j `from` j'' `(`l`,`d`))` in the `inBuffer` of object j must equal the data value d' in the message `(to` j' `from` j'' `(`l'`,`d'`))` in the `inBuffer` of object j'.

For example, an initial state corresponding to the synchronous machine in Fig. 2 could be the following:

```
{< 1 : C1 | clock : c1(t0),  roundTimer : T − c1(t0),
            outputBackoffTimer : INF,
            inBuffer : (to 1 from 3 (1, d^3_o1))
```

```
                    (to 1 from e (2, d^e_o1)),
          outBuffer : none,  state : s_1,
          localWiring : 1 --> 3.1 >
< 2 : C_2 | clock : c_2(t_0),  roundTimer : T - c_2(t_0),
          outputBackoffTimer : INF,
          inBuffer : (to 2 from 3 (1, d^3_o3))
                    (to 2 from e (2, d^e_o2)),
          outBuffer : none,  state : s_2,
          localWiring : 1 --> e.3 >
< 3 : C_3 | clock : c_3(t_0),  roundTimer : T - c_3(t_0),
          outputBackoffTimer : INF,
          inBuffer : (to 3 from 1 (1, d^1_o1))
                    (to 3 from e (2, d^e_o1)),
          outBuffer : none, state : s_3,
          localWiring : 1 --> 1.1 ; 1 --> e.1 ;
                        2 --> e.2 ; 3 --> 2.1 >
< e : Env | clock : c_e(t_0),  roundTimer : T - c_e(t_0),
           outputBackoffTimer : INF,
           inBuffer : (to e from 3 (1, d^3_o1))
                     (to e from 3 (2, d^3_o2))
                     (to e from 2 (3, d^2_o1)),
           outBuffer : none,
           localWiring : 1 --> 1.2 ; 1 --> 3.2 ;  2 --> 2.2 >
    ;
t_0 }
```

for values $s_1, d^1_{o_1}, \ldots$ of appropriate sorts.

5 Consequences of Clock Synchronization

The following, perhaps slightly surprising, facts are two consequences of having local clock functions that are only piecewise continuous. They do not apply when the clock functions are continuous.

Fact 1. *Although the clock synchronization ensures that the difference between any local clock and the global time is always* strictly less *than ϵ, it may happen that a timer that is set to expire at (local) time t expires* exactly *ϵ time "units" later; that is, the timer could expire at global time $t + \epsilon$.*

Proof. We can have a clock $c(t)$ obeying $|c(t) - t| < \epsilon$ and such that it has a discontinuity at global time 11 but gets arbitrarily close to the value 10 for $t < 11$, i.e., $\lim_{t<11,\, t\to 11} c(t) = 10$. Instead at time 11, the clock jumps to a value $10 < c(11) < 12$.

The above fact has a practical consequence in that timer-driven events that are supposed to happen at ("local") time T may happen at any global time in the interval $(T - \epsilon, T + \epsilon]$ (instead of in the right-open interval $(T - \epsilon, T + \epsilon)$).

Note that if all clocks are *continuous*, this behavior is impossible so that a timer set to expire at local time t must indeed expire within the global time interval $(t-\epsilon, t+\epsilon)$.

Fact 2. *In the presence of only piecewise continuous clock with maximum drift strictly less than ϵ, a periodic timer of period p reset when the local timer expires can drift from the ideal time strictly beyond ϵ.*

Proof. It is sufficient to show a concrete example. Let $\epsilon = 1$, $p = 10$, and $c(0) = 0$. As shown by Fact 1, we can have $\lim_{t<11,\, t\to 11} c(t) = 10$ but $c(11) = 11.9$. Then the timer is reset to 10, and we can have $c(21.9) = 22.8$. Therefore, the second time the timer is set, namely at time 21.9, the time has drifted 1.9 time units from the ideal timer. Note that, by repeating this type of behavior, it is possible for the timer to drift an arbitrary distance from the ideal timer.

This second fact implies that we must set the timers not to the round time T, but must adjust the new timer value to account for the possible "clock jump". Fortunately, in our asynchronous model, this "offset" can be computed without knowing anything except the local clock value, the length of the period, and the "starting time" of the first period:

Fact 3. *It is possible to keep a timer with a local clock $c(t)$ set to expire at times $k \cdot p + \tau$ (with $2\epsilon \leq p$) with a drift less than or equal to ϵ from the ideal timer by resetting it each time it expires, say at time $c(t)$, to the new value $p - \texttt{adjust}(c(t), p, \tau)$, where $\texttt{adjust}(c(t), p, \tau) = c(t) - ((\lfloor (c(t) - \tau)/p \rfloor * p) + \tau)$.*

Therefore, given the current local time t, the round time T, and the start time of the timer in the first round (τ), the value which should be subtracted from the new timer value is $\texttt{adjust}(t, T, \tau)$.

This need to explicitly incorporate into the model the adjustment for clock resets when the timers expire is not due to peculiarities with our way of modeling clock synchronization, but must be done for the system whenever we have a clock synchronization algorithm that can adjust the clocks in "jumps". Of course, if the clock functions are not only monotonic but also *continuous* (and not just piecewise continuous), this adjustment is not needed, because no such "jumps" can occur.

6 Correctness of the PALS Transformation

Given a typed machine ensemble $\mathcal{E}$ with associated environment constraint c_e, and values $\alpha_{min}, \alpha_{max}, \mu_{min}, \mu_{max}, T, \epsilon$, and a vector $\boldsymbol{c}$ of ϵ-drift clock functions, we have defined in Section 4 its asynchronous PALS transformation $\mathcal{A}(\mathcal{E}_{c_e}, \alpha_{min}, \alpha_{max}, \mu_{min}, \mu_{max}, T, \epsilon, \boldsymbol{c})$ as an object-oriented Real-Time Maude model. This section establishes a precise relationship between the synchronous composition $\mathcal{M}_{\mathcal{E}}$ of the ensemble $\mathcal{E}$ defined in Section 3 and the asynchronous model $\mathcal{A}(\mathcal{E}_{c_e}, \alpha_{min}, \alpha_{max}, \mu_{min}, \mu_{max}, T, \epsilon, \boldsymbol{c})$.

Notation. When the various parameters of the PALS transformation are implicit, we sometimes write $\mathcal{A}(\mathcal{E})$ for $\mathcal{A}(\mathcal{E}_{c_e}, \alpha_{min}, \alpha_{max}, \mu_{min}, \mu_{max}, T, \epsilon, \boldsymbol{c})$.

Each synchronous transition step in the synchronous composition $M_{\mathcal{E}}$ corresponds to multiple rewrite steps in $\mathcal{A}(\mathcal{E})$. The key idea is to define "larger" transition steps that consist of multiple rewrite steps in $\mathcal{A}(\mathcal{E})$, so that each of these large transitions corresponds to a single transition step in $M_{\mathcal{E}}$.

Definition 6. *A state* $\{C\,;t\}$ *in* $\mathcal{A}(\mathcal{E})$ *is called* stable *iff*

- *all input buffers in* C *are full,*
- *all output buffers are empty (*`none`*), and*
- *there are no messages "in transit" in* C*.*

Intuitively, a stable state corresponds to a state $(\boldsymbol{s}, \boldsymbol{i})$ in $M_{\mathcal{E}}$, and a sequence of rewrite steps between two stable states corresponds to a transition in $M_{\mathcal{E}}$. However, due to time ticks, there could be a rewrite sequence from one stable state to another (very similar) stable state that does not correspond to a transition in $M_{\mathcal{E}}$.

6.1 The Transition System of Stable Configurations

Definition 7. *A* behavior *in* $\mathcal{A}(\mathcal{E})$ *is a sequence*

$$\{C_0\,;\,t_0\} \longrightarrow \{C_1\,;\,t_1\} \longrightarrow \{C_2\,;\,t_2\} \longrightarrow \cdots$$

of rewrite steps in $\mathcal{A}(\mathcal{E})$*, where* $\{C_0\,;\,t_0\}$ *is an initial state of the form described in Section 4.9.*

Definition 8. *Let Stable*$(\mathcal{A}(\mathcal{E}))$ *denote both the set of states and the transition system whose states are the stable states of* $\mathcal{A}(\mathcal{E})$*, and where the transitions, denoted*

$$\{C\,;\,t\} \longrightarrow_{st} \{C'\,;\,t'\}$$

are defined as follows: $\{C;t\} \longrightarrow_{st} \{C';t'\}$ *iff there exists a behavior of* $\mathcal{A}(\mathcal{E})$ *of the form*

$$\{C_0\,;\,t_0\} \longrightarrow \{C_1\,;\,t_1\} \longrightarrow \cdots \longrightarrow \{C_n\,;\,t_n\} \longrightarrow \{C_{n+1}\,;\,t_{n+1}\} \longrightarrow \cdots$$

and numbers k, k' *with* $k < k'$ *such that*

- $C = C_k$ *and* $C' = C_{k'}$ *are stable configurations,* $t = t_k$ *and* $t' = t_{k'}$*, and*
- $\{C_k\,;\,t_k\} \longrightarrow \{C_{k+1}\,;\,t_{k+1}\} \longrightarrow \cdots \longrightarrow \{C_{k'}\,;\,t_{k'}\}$ *is a subsequence of rewrites in such a behavior such that*
  - *the sequence contains at least one application of an instantaneous rewrite rule, and*
  - *if* C_j *is* not *a stable state, for* $k < j < k'$*, then there is no* $j < l < k'$ *such that* C_l *is a stable state.*

6.2 Relating the Synchronous and the Asynchronous Models

In this section we prove that $Stable(\mathcal{A}(\mathcal{E}))$ and $M_{\mathcal{E}}$ are bisimilar by first relating stable states to states in the synchronous ensemble composition, and then by showing that each synchronous transition has a corresponding stable transition, and vice versa.

The relation between a stable state and the corresponding state in the synchronous composition is fairly obvious:

Definition 9. *Let*

$$sync : Stable(\mathcal{A}(\mathcal{E})) \rightarrow S^{\mathcal{E}} \times D_i^{\mathcal{E}}$$

be a function that maps each stable state of the asynchronous model to the corresponding state of the synchronous system as follows:

- *The local state of each object j, given in the object's* `state` *attribute, determines the local state in M_j. This is well defined, since in a* stable *state, the* `state` *attribute does not have a "delayed" value.*
- *The messages in the input buffers determine the state of the environment input and feedback wires using the functions $fout_j$, $j \in J$ defined in Definition 3 of Section 3.*

More precisely, $sync(\{C; t\})$ is defined to be the tuple

$$(((s_1, \ldots, s_j), ((d^1_{o_1}, \ldots, d^1_{o_{m_1}}), \ldots, (d^j_{o_1}, \ldots, d^j_{o_{m_j}}))),\ (d^e_{o_1}, \ldots, d^e_{o_{m_e}}))$$

where:

- *the* `state` *attribute of the object i has the value s_i in C,*
- *d^i_k (for $i \neq e$) equals $*$ iff for all "connections" of the form k* `-->` *$o.p$ (for the given k) in the* `localWiring` *attribute of object i in C, the object o is the identifier of the environment object of class* `Env`*, and*
- *otherwise, d^i_k, for $i \in J \cup \{e\}$, equals the value d in a message*

  `to` *l* `from` *i* `(`*p*`,` *d*`)`

  in the `inBuffer` *of object l in C for some l, p for which the* `localWiring` *attribute of object i contains a connection k* `-->`*l* `.`*p.*

The requirement that the messages in the `inBuffer`*s in the initial states are "wiring consistent" (see Section 4.9) ensures that sync is well-defined for all reachable stable states.*

Example 2. Let t_0 be the initial state given in Section 4.9. Then $sync(t_0)$ is the machine ensemble in Fig. 2, where the internal state of machine M_i is s_i, where the value in the feedback wire from M_1 to M_3 is $d^1_{o_1}$, and so on. That is, $sync(t_o)$ is the tuple

$$(((s_1, s_2, s_3), (d^1_{o_1}, *, (d^3_{o_1}, *, d^3_{o_3}))),\ (d^e_{o_1}, d^e_{o_2})).$$

The main result we want is the following:

Theorem 1. *sync defines a bisimulation between the transition systems* $Stable(\mathcal{A}(\mathcal{E}))$ *and* $(S_{M_\mathcal{E}} \times D_i^\mathcal{E}, \longrightarrow_{M_\mathcal{E}})$.

Proof. We need to prove:

1. If $(\boldsymbol{s}, \boldsymbol{i}) \longrightarrow_{M_\mathcal{E}} (\boldsymbol{s'}, \boldsymbol{i'})$, then, for each stable state c such that $sync(c) = (\boldsymbol{s}, \boldsymbol{i})$, there must exist a transition $c \longrightarrow_{st} c'$ of stable states c, c' such that $sync(c') = (\boldsymbol{s'}, \boldsymbol{i'})$.
2. If $c \longrightarrow_{st} c'$, then it must be the case that $sync(c) \longrightarrow_{M_\mathcal{E}} sync(c')$.

These two properties are proved as, respectively, Lemmas 9 and 8. □

Furthermore, a CTL^* formula holds in the associated Kripke structure $(S^\mathcal{E} \times D_i^\mathcal{E}, \longrightarrow_{M_\mathcal{E}}, L)$ if and only if the formula holds in the corresponding Kripke structure $(Stable(\mathcal{A}(\mathcal{E})), \longrightarrow_{st}, sync; L)$ associated with $Stable(\mathcal{A}(\mathcal{E}))$:

Theorem 2. *For any formula* $\phi \in CTL^*(AP)$ *and for any stable initial configuration* $\{C_0; t_0\}$ *of the form described in Section 4.9, we have*

$$\begin{array}{c}(Stable(\mathcal{A}(\mathcal{E})), \longrightarrow_{st}, (sync; L)), \{C_0\,;\, t_0\} \models \phi \\ \Updownarrow \\ (S^\mathcal{E} \times D_i^\mathcal{E}, \longrightarrow_{M_\mathcal{E}}, L), sync(\{C_0\,;\, t_0\}) \models \phi.\end{array}$$

Proof. The function *sync* defines not only a bisimulation between transition systems, but also one between the Kripke structures $(Stable(\mathcal{A}(\mathcal{E})), \longrightarrow_{st}, sync; L)$ and $(S^\mathcal{E} \times D_i^\mathcal{E}, \longrightarrow_{M_\mathcal{E}}, L)$, since if $sync(\{C\,;\, t\}) = s$, then $\{C\,;\, t\}$ and s satisfy the same atomic propositions in their respective Kripke structures, since $(sync; L)(\{C\,;\, t\}) = L(sync(\{C\,;\, t\})) = L(s)$. It is well-known that bisimilar structures satisfy the same CTL^* formulas (see, e.g., [2]). □

7 Lemmas for the Bisimulation Theorem

Lemma 1. *Let a timer in an object have value* q *at time* t_0*. Then, the timer expires at some global time in the interval* $(q + c(t_0) - \epsilon,\ q + c(t_0) + \epsilon]$ *for* c *the local* ϵ*-drift clock function of the object.*

Proof. Some helpful lemmas, such as that the effect of two applications of the function `delta` equals one `delta` application for the sum of the time advances, etc., are proved below.

At global time t, the timer has the value q `monus` $(c(t) - c(t_0))$ if everything is done using *one* application of the `delta` function, which we will argue below in Lemma 2 can be assumed. The timer will expire the first time that the above value reaches 0. This happens at the first global time t_1 when $c(t_1) - c(t_0) \geq q$, which is the same as $c(t_1) \geq q + c(t_0)$. This may happen at time t_1 either when

1. $c(t_1) = q + c(t_0)$, or when

2. $c(t_1) > q + c(t_0)$ and there is no $t' < t_1$ such that the timer expires at time t'.

Let us consider case (1) first. By definition of drift functions, t_1 is in the global time interval $(c(t_1) - \epsilon,\ c(t_1) + \epsilon)$, which in case (1) equals the interval $(q + c(t_0) - \epsilon,\ q + c(t_0) + \epsilon)$, which is inside the interval in the theorem.

Let us now consider case (2). Now, $c(t_1) > q + c(t_0)$, and hence we have $c(t_1) - \epsilon > q + c(t_0) - \epsilon$. In addition, due to the assumption of ϵ-drift functions, we have $t_1 > c(t_1) - \epsilon$. Together, these last inequalities give $t_1 > c(t_1) - \epsilon > q + c(t_0) - \epsilon$, which proves the lower bound in the interval in the theorem. Furthermore, t_1 cannot be greater than $q + c(t_0) + \epsilon$, because at time $q + c(t_0) + \epsilon$, the clock must be at least $q + c(t_0)$ and the timer would expire then. This proves the upper bound. □

In the above proof, we reasoned about the expiration of a timer given that the tick rule and the `delta` function are only applied once. We notice that the timer value is only changed by either the tick rule, or when the timer expires. The following shows that we can contract multiple tick steps into one for the sake of reasoning about timers:

Lemma 2. *For any object in state o, we have*

$$\texttt{delta}(\texttt{delta}(o, t_0, \Delta), t_0 + \Delta, \Delta') = \texttt{delta}(o, t_0, \Delta + \Delta')$$

for any time values t_0, Δ, Δ'.

Proof. The proof of this lemma follows directly from the equations defining the semantics of the `delta` function in Section 4.8. Mathematically, this just follows from the general fact that those equations imply that `delta` is an *action* on the additive monoid of time $(\mathbb{R}_{\geq 0}, +, 0)$ over the configurations of objects and messages. The details are left to the reader. □

Lemma 3. *The* `roundTimer` *for each object expires somewhere in the global time interval $(i \cdot T - \epsilon,\ i \cdot T + \epsilon]$ for all $i \in \mathbb{N}$, from any initial state of the form assumed in Section 4.9.*

Proof. In the initial state, the value of `roundTimer` for object j is $T - c_j(T - \epsilon)$, and the local clock is $c_j(T - \epsilon)$. It follows directly from Lemma 1 that the timer expires somewhere in the global time interval $(T - \epsilon,\ T + \epsilon]$. In addition, it follows trivially that the local clock is greater than or equal to T.

Assume that at the time when the `roundTimer` first expires, the *local* clock is $T + \Delta$ for some $2\epsilon > \Delta \geq 0$ (where Δ will be strictly greater than 0 only if the timer expired in a "clock jump"). In the rules `applyTrans`, the `roundTimer` is reset to $T - ((T + \Delta) - \lfloor \frac{T+\Delta}{T} \rfloor \cdot T)$, which equals $T - \Delta$, since T is greater than 2ϵ according to our constraints. The sum of the local clock and the newly set timer is therefore $2 \cdot T$, and the timer will therefore expire the next time sometime in the global time interval $(2T - \epsilon,\ 2T + \epsilon]$ according to Lemma 1. This reasoning can be replicated for any round i. □

Lemma 4. *The* `outputBackoffTimer` *of each object in states reachable from the initial states of the given form expires somewhere in the global time interval* $((i \cdot T) + (2\epsilon \texttt{ monus } \mu_{min}) - \epsilon,\ (i \cdot T) + (2\epsilon \texttt{ monus } \mu_{min}) + \epsilon]$ *for each* $i \in \mathbb{N}$ *with* $i \geq 1$.

Proof. In the initial state, the `outputBackoffTimer`s are turned off. This timer is set in the rule `applyTrans` and, for the environment, rule `consumeInputAndGenerateOutput`, when the `roundTimer` expires. According to the proof of Lemma 3, the local clock is $T + \Delta$ the first time this happens (for Δ as described in the above proof of Lemma 3). In these rules, the `outputBackoffTimer` is set to $(2\epsilon \texttt{ monus } \mu_{min}) \texttt{ monus } \Delta$. We need to consider three cases: (1) $2\epsilon \leq \mu_{min}$, (2) $2\epsilon > \mu_{min}$ and $(2\epsilon - \mu_{min}) < \Delta$, and (3) $2\epsilon > \mu_{min}$ and $(2\epsilon - \mu_{min}) \geq \Delta$. In case (1), the `outputBackoffTimer` expires when it is set, which according to Lemma 3 takes place in the global time interval $(i \cdot T - \epsilon,\ i \cdot T + \epsilon]$, which equals the time interval in Lemma 4 when $2\epsilon \leq \mu_{min}$. In case (2), the `outputBackoffTimer` is also set to 0, but the desired time interval in Lemma 4 is now $((i \cdot T) + (2\epsilon - \mu_{min}) - \epsilon,\ (i \cdot T) + (2\epsilon - \mu_{min}) + \epsilon]$. According to Lemma 1, the timer can expire no earlier than at time $T + \Delta - \epsilon$, which is later than or equal to the lower bound $T + (2\epsilon - \mu_{min}) - \epsilon$ in Lemma 4, since case (2) assumes $(2\epsilon - \mu_{min}) < \Delta$. The upper bound $T + (2\epsilon - \mu_{min}) + \epsilon$ follows from Lemma 3, since the timer expires at latest at time $T + \epsilon$, and we have assumed that in case (2) that $(2\epsilon - \mu_{min}) > 0$. Finally, for case (3), the *local* clock of the object under consideration is $T + \Delta$, and the `outputBackoffTimer` is set to $(2\epsilon - \mu_{min}) - \Delta \geq 0$. It then follows from Lemma 1 that this local `outputBackoffTimer` expires for the first time somewhere in the global time interval $(T + (2\epsilon - \mu_{min}) - \epsilon,\ T + (2\epsilon - \mu_{min}) + \epsilon]$. Again, this reasoning can be replicated for any round i. □

Remark. The above lemmas should be read as safety and not as liveness guarantees. That is, if time advances at all that far, then the timers expire in the given intervals.

Lemma 5. *Messages are sent from the output buffers in the global time interval* $(iT - \epsilon + \max(2\epsilon - \mu_{min}, \alpha_{min}),\ iT + \epsilon + \max(2\epsilon - \mu_{min}, \alpha_{max})]$ *during round each round* $i \geq 1$.

Proof. Messages are only generated into the output buffers when the `roundTimer`s expire. Then, the `outputBackoffTimer`s are set, and expire within the time intervals given in Lemma 4. At the same time (that is, when the `roundTimer` expires), the execution delay is set to somewhere between α_{min} and α_{max}. Messages are only sent when the output backoff timer has expired and when the execution delay has elapsed. Furthermore, we see that the backoff timer is only turned off when it has expired (that is, when it has value 0). From Lemma 4, the backoff timer expires *strictly later* than global time $iT - \epsilon + (2\epsilon \texttt{ monus } \mu_{min})$; furthermore, since the `roundTimer` expires strictly later than global time $iT - \epsilon$, the execution delay ends strictly later than at global time $iT - \epsilon + \alpha_{min}$, together giving the lower bound in the lemma, since $\max(2\epsilon \texttt{ monus } \mu_{min}, \alpha_{max}) = \max(2\epsilon - \mu_{min}, \alpha_{max})$ since $\alpha_{max} \geq 0$.

As for the upper bound, the `roundTimer` expires at latest at time $iT+\epsilon$, and hence the messages are ready to be sent at latest at global time $iT+\epsilon+\alpha_{max}$. Likewise, the latest time the backoff expires is at time $iT+\epsilon+(2\epsilon \texttt{ monus } \mu_{min})$, together giving the upper bound. □

We continue our timeline proof by showing that the messages generated in round i are received at the appropriate times:

Lemma 6. *The messages sent into the configuration in round i will be received at times within the global time interval* $(i \cdot T + \epsilon,\ (i+1) \cdot T - \epsilon]$.

Proof. As seen in Lemma 5, each message is sent out in round i somewhere in the global time interval $(iT-\epsilon+\max(2\epsilon-\mu_{min}),\ iT+\epsilon+\max(2\epsilon-\mu_{min},\alpha_{max})]$, and is given a delay between μ_{min} and μ_{max}. We see that the remaining delay of a message decreases by the same amount that global time advances, and that $\texttt{mte}(m) = 0$ for an undelayed message (which by the identity attribute of the message delay operator is the same as a message with delay 0) implies that the message must be received when its delay reaches 0 "for the first time." Therefore, each of these messages is created in the interval $(iT-\epsilon+\max(2\epsilon-\mu_{min},\alpha_{min}),\ iT+\epsilon+\max(2\epsilon-\mu_{min},\alpha_{max})]$ and is received at some time in the global interval $(iT-\epsilon+\max(2\epsilon-\mu_{min},\alpha_{min})+\mu_{min},\ iT+\epsilon+\max(2\epsilon-\mu_{min},\alpha_{max})+\mu_{max}]$.

To prove the lemma, we must therefore show

1. $(i \cdot T)+\epsilon \leq iT-\epsilon+\max(2\epsilon-\mu_{min},\alpha_{min})+\mu_{min}$ and
2. $iT+\epsilon+\max(2\epsilon-\mu_{min},\alpha_{max})+\mu_{max} \leq ((i+1)\cdot T)-\epsilon$.

Requirement (1) follows by arithmetic. The upper time limit requirement (2) reduces to proving $\epsilon+\max(2\epsilon-\mu_{min},\alpha_{max})+\mu_{max} \leq T-\epsilon$, which follows from the global requirement that $T \geq \mu_{max}+2\epsilon+\max(2\epsilon-\mu_{min},\alpha_{max})$. □

These theorems, together with a trivial inspection of the rules and the well-known timed behavior, together prove the time line described in Section 4.3:

Lemma 7. *For all (time diverging [6]) behaviors from the given initial states, the behaviors have the following time line for all* $i \in \mathbb{N}$*:*

- *at times in the global time interval* $(iT-\epsilon,\ iT+\epsilon]$ *the* `roundTimer`*s expire, all input buffers are read, and the transitions corresponding to those in the synchronous system are applied. The resulting states and output messages (stored in the output buffers) are undelayed no later than at global time* $iT+\epsilon+\alpha$*, and the backoff timer expires at latest at time* $iT+\epsilon+(2\epsilon \texttt{ monus } \mu_{min})$*, ensuring that all these messages are sent to the global configuration in the global time interval* $(iT-\epsilon+\max(2\epsilon-\mu_{min},\alpha_{min}),\ iT+\epsilon+\max(2\epsilon-\mu_{min},\alpha_{max})]$*; furthermore, also the messages generated nondeterministically by the environment are sent into the global configuration in this interval;*

- *these messages are received at times within the global time interval* $(i \cdot T + \epsilon,\ (i+1) \cdot T - \epsilon]$*, ensuring that all messages are received and stored in the respective input buffers before or at global time* $(i+1) \cdot T - \epsilon$*;*
- *a new round therefore begins at times in the global time interval* $((i+1) \cdot T - \epsilon,\ (i+1) \cdot T + \epsilon]$*: the* `roundTimer`*s expire, all input buffers are read, and the transitions corresponding to those in the synchronous system are applied, and so on.* □

The above lemma defines the behaviors from the given initial states, and are crucial in the proof of the following:

Lemma 8. *Let* $\{C\,;\,t\} \longrightarrow_{st} \{C'\,;\,t'\}$ *be a transition in Stable*$(\mathcal{A}(\mathcal{E}))$ *for a machine ensemble* $\mathcal{E}$*. Then there is a transition* $(\boldsymbol{s}, \boldsymbol{i}) \longrightarrow_{M_\mathcal{E}} (\boldsymbol{s'}, \boldsymbol{i'})$ *such that* $synch(\{C\,;\,t\}) = (\boldsymbol{s}, \boldsymbol{i})$ *and* $synch(\{C'\,;\,t'\}) = (\boldsymbol{s'}, \boldsymbol{i'})$*.*

Proof. (Sketch) Since C is a stable configuration, all its input buffers are full, all the output buffers are empty, and there are no (delayed or undelayed) messages in C outside of these buffers. In addition, the backoff timers are turned off. It is easy to see by inspecting the rules that only the tick rule, the rule `applyTrans`, or the rule `consumeInputAndGenerateOutput` can be applied. Repeated applications of the tick rule leave us in stable states, until eventually the `roundTimer`s start expiring. At these times, the rules `applyTrans` and `consumeInputAndGenerateOutput` generates new messages and delayed states, and by the above time line, all these messages will reach the input buffers before the `roundTimers` expire, hence we have reached a new stable state C' when all input buffers are full, but before transitions for the "next" round are applied.

The above reasoning shows that from a stable state $\{C\,;\,t\}$ we can always reach another stable state $\{C'\,;\,t'\}$ by a transition $\{C\,;\,t\} \longrightarrow_{st} \{C'\,;\,t'\}$.

Now, we must show that such a stable transition $\{C\,;\,t\} \longrightarrow_{st} \{C'\,;\,t'\}$ actually corresponds to a transition $synch(\{C\,;\,t\}) \longrightarrow_{M_\mathcal{E}} synch(\{C'\,;\,t'\})$ in $M_\mathcal{E}$. In particular, assume that $synch(\{C;t\} = ((\boldsymbol{s}, (\boldsymbol{d^1}, \ldots, \boldsymbol{d^{|J|}})), \boldsymbol{e})$ and that $synch(\{C';t'\}) = ((\boldsymbol{s'}, ((\boldsymbol{d^1})', \ldots, (\boldsymbol{d^{|J|}})')), \boldsymbol{e'})$. Then, we must prove that $((\boldsymbol{s}, (\boldsymbol{d^1}, \ldots, \boldsymbol{d^{|J|}})), \boldsymbol{e}) \longrightarrow_\mathcal{E} ((\boldsymbol{s'}, ((\boldsymbol{d^1})', \ldots, (\boldsymbol{d^{|J|}})')), \boldsymbol{e'})$; that is,

1. $(\boldsymbol{s'}, ((\boldsymbol{d^1})', \ldots, (\boldsymbol{d^{|J|}})')) = \pi_1(\delta_\mathcal{E}((\boldsymbol{s}, (\boldsymbol{d^1}, \ldots, \boldsymbol{d^{|J|}})), \boldsymbol{e}))$, and
2. $c_e(\boldsymbol{e'})$.

The second requirement is immediate, as the rule `consumeInputAndGenerateOutput` only generates input from the environment that satisfies c_e.

For the first requirement, we must therefore prove that $\pi_1(\delta_\mathcal{E}(\boldsymbol{e}, \pi_1(synch(\{C\,;t\}))))$ is equal to $\pi_1(synch(\{C'\,;t'\}))$.

Consider $\delta_\mathcal{E}(\boldsymbol{e}, \pi_1(synch(\{C\,;\,t\})))$. After this (synchronous) transition has been taken, the resulting "internal" state s'_l in machine l is given by

$$s'_l = \pi_1(\delta_{M_l}(in_l(\boldsymbol{e}, \pi_2(\pi_1(synch(\{C\,;\,t\})))), s_l))$$

for s_l the current internal state of machine l (this s_l can be extracted from $synch(\{C\,;\,t\})$). The kth element in $in_l(\boldsymbol{d}, \pi_2(\pi_1(synch(\{C\,;\,t\}))))$ is defined to

be d_i if $src(l,k) = (e,i)$, and is the defined to be the ith element of the jth component of $\pi_2(\pi_1(synch(\{C\,;\,t\})))$ if $src(l,k) = (j,i)$. By Definition 9, this latter element equals the data v in the message `(to` l `from` j `(`k`,`v`))` in the `outBuffer` of object j in C.

Now, consider the rule `applyTrans` for object l for the messages in C and those created by the environment. After applying the rule, the resulting `state` attribute will be $\pi_1(\delta_{M_l}(vect_{[l]}(B), s_l))$ (after the delay on the result has disappeared). By definition, the kth element of $vect_{[l]}(B)$ has the value v if the object just read a message `(to` l `from` j' `(`k`,`v`))` from its input buffer. This means that the message must also have been in j''s `outBuffer` in C, and is only sent to l if $src(l,k) = (j',r)$ for some r. By assumption, $src(l,k) = (j,i)$, hence this message is the same as the above, as we assume consistent messages in the output buffers (the function `makeMsg` is not further defined).

We have now shown that the resulting states in C' and $\delta_{\mathcal{E}}(\boldsymbol{e}, \pi_1(synch(\{C\,;\,t\})))$ are the same.

It is equally straight-forward, although somewhat tedious, to prove that the values in the resulting feedback wires correspond to the generated messages in the round. □

Lemma 9. *Let* $((\boldsymbol{s}, (\boldsymbol{d^1}, \ldots, \boldsymbol{d}^{|J|})),\ \boldsymbol{e}) \longrightarrow_{\mathcal{E}} ((\boldsymbol{s'}, ((\boldsymbol{d^1})', \ldots, (\boldsymbol{d}^{|J|})')),\ \boldsymbol{e'})$ *be a transition in* $M_{\mathcal{E}}$ *for a machine ensemble* $\mathcal{E}$*. Then, for all* $\{C\,;\,t\}$ *such that* $synch(\{C\,;\,t\}) = ((\boldsymbol{s}, (\boldsymbol{d^1}, \ldots, \boldsymbol{d}^{|J|})),\ \boldsymbol{e})$*, there exists a* $\{C'\,;\,t'\}$ *such that* $synch(\{C'\,;\,t'\}) = ((\boldsymbol{s'}, ((\boldsymbol{d^1})', \ldots, (\boldsymbol{d}^{|J|})')),\ \boldsymbol{e'})$ *and* $\{C\,;\,t\} \longrightarrow_{st} \{C'\,;\,t'\}$.

Proof. (Sketch) The new environment output $\boldsymbol{e'}$ can be generated by the environment object as long as it as valid output. For any stable state C whose feedback wire states and internal states define $(\boldsymbol{s}, (\boldsymbol{d^1}, \ldots, \boldsymbol{d}^{|J|}))$, the time-line reasoning then implies that these messages will all be read in the correct time interval, and the next feedback states will be generated. The remaining details are left to the reader. □

8 Optimality Results

This section shows that the period $T = 2\epsilon + \mu_{max} + \max(\alpha_{max}, 2\epsilon - \mu_{min})$ is the smallest possible for PALS.

Proposition 1. *Assume that each object reads its input buffer at its* local *time* t_0*, and at that time starts performing a transition and generating new messages. To ensure that all such generated messages are read by all other objects at or before their* local *times* $t_0 + T$*, we must have*

$$T \geq 2\epsilon + \mu_{max} + \alpha_{max}.$$

Proof. Assume for a proof by contradiction that T is *strictly smaller* than $2\epsilon + \mu_{max} + \alpha_{max}$. Then, $T = 2\epsilon + \mu_{max} + \alpha_{max} - \Delta$ for some $\Delta > 0$. Furthermore, let k be a number $k \geq 3$ such that $\Delta < k \cdot \epsilon$.

Now, assume two objects with local clocks c_1 and c_2 such that $c_1(t_0+\epsilon-\frac{\Delta}{k}) = t_0$ and $c_2(t_0+T-\epsilon+\frac{\Delta}{k}) = t_0+T$. That is, at *global* time $t_0+\epsilon-\frac{\Delta}{k}$, the object 1 generates messages that should arrive no later than at *global* time $t_0+T-\epsilon+\frac{\Delta}{k}$, when object 2 reads messages for the next round. However, it is easy to see that, in the worst case (longest execution time and network delay), the messages from object 1 arrive at *global* time $t_0+\epsilon-\frac{\Delta}{k}+\mu_{max}+\alpha_{max}$, which is strictly later than global time $t_0+T-\epsilon+\frac{\Delta}{k} = t_0+\epsilon+\mu_{max}+\alpha_{max}-\frac{(k-1)\Delta}{k}$, since $\frac{(k-1)\,\Delta}{k} > \frac{\Delta}{k}$ for $k \geq 3$, so that object 2 misses the messages sent from object 1.

Proposition 1 proves optimality of T when $\alpha_{max} \geq 2\epsilon - \mu_{min}$. For the converse, and highly unlikely, case where $2\epsilon - \mu_{min} > \alpha_{max}$, it is harder to claim a "general" optimality result. PALS uses a backoff timer to avoid that messages arrive too early. One could imagine variations of PALS where there were no such backoff timers are used, but where messages are instead equipped with, e.g., sequence numbers denoting the round in which they were generated. In such cases, a backoff timer would not be needed, and $T \geq 2\epsilon + \mu_{max} + \alpha_{max}$ might suffice as a smallest period.

However, if we want to ensure that each message arrives in the right round by using backoff timers, then the backoff timers must be set to at least $2\epsilon-\mu_{min}$:

Proposition 2. *To ensure that a message generated in round i (i.e., at* local *time $i \cdot T$) does not arrive too early (i.e., is read by another object at that object's local time $i \cdot T$), the message must not be sent before* local *time $i \cdot T + 2\epsilon - \mu_{min}$.*

Proof. Assume that object 1 sends a message to object 2 *before* local time $i \cdot T + 2\epsilon - \mu_{min}$. That is, it sends the messages at its *local* time $i \cdot T + 2\epsilon - \mu_{min} - \Delta$ for some $\Delta > 0$. Again, we let $k \geq 3$ be some number such that $\Delta < k \cdot \epsilon$.

Furthermore, suppose that $c_1(i \cdot T + 2\epsilon - \epsilon + \frac{\Delta}{k} - \mu_{min} - \Delta) = i \cdot T + 2\epsilon - \mu_{min} - \Delta$. That is, the messages from object 1 are sent at *global* time $i \cdot T + 2\epsilon - \epsilon + \frac{\Delta}{k} - \mu_{min} - \Delta$. With the smallest possible network delay, these messages may arrive at *global* time $i \cdot T + \epsilon - \frac{(k-1)\Delta}{k}$, whereas it could be the case that $c_2(i \cdot T + \epsilon - \frac{\Delta}{k}) = i \cdot T$, and, therefore, object 2 would read the messages from object 1 one round too early.

The optimality of the period follows immediately:

Proposition 3. *If each message for round $i+1$ is sent no earlier than at* local *time $i \cdot T + 2\epsilon - \mu_{min}$, then we must have*

$$T \geq 4\epsilon + \mu_{max} - \mu_{min}$$

to ensure that each object has received these messages at its local *time $i \cdot T + T$.*

Proof. Assume for a counterexample that $T = 4\epsilon + \mu_{max} - \mu_{min} - \Delta$ for $\Delta > 0$ with $\Delta < k \cdot \epsilon$ for some $k \geq 3$.

Let us assume two objects with local clocks c_1 and c_2, where $c_1(i \cdot T + 2\epsilon - \mu_{min} + (\epsilon - \frac{\Delta}{k})) = i \cdot T + 2\epsilon - \mu_{min}$. That is, object 1 does not send its messages

for round $i+1$ earlier than at *global* time $i \cdot T + 2\epsilon - \mu_{min} + (\epsilon - \frac{\Delta}{k})$. In the worst case, these messages arrive at *global* time $i \cdot T + 3\epsilon - \mu_{min} + \mu_{max} - \frac{\Delta}{k}$. However, it could well be the case that $c_2(i \cdot T + T - \epsilon + \frac{\Delta}{k}) = i \cdot T + T$. That is, the messages arriving at time $i \cdot T + 3\epsilon - \mu_{min} + \mu_{max} - \frac{\Delta}{k}$ arrive later than the global time $i \cdot T + T - \epsilon + \frac{\Delta}{k} = i \cdot T + 3\epsilon + \mu_{max} - \mu_{min} - \frac{(k-1)\Delta}{k}$ when object 2 reads its messages for round $i+1$.

9 Conclusions

This work has presented a formal mathematical model of, and formal requirements for legal instances of, the generic PALS architectural pattern for obtaining correct-by-construction real-time systems. Based on the PALS formal model, we have given proofs of correctness of PALS, and of optimality of the PALS period. We have also given formal executable specifications of the wrappers used to build PALS as a collection of wrapped abstract machines communicating through message passing. This latter, executable aspect is of practical importance as a basis on which correct-by-construction PALS implementations could be developed by code generation, or, more precisely, by code "weaving."

We believe that PALS as a complexity-reducing formalized architectural pattern can greatly increase system quality and can greatly reduce the cost of design, implementation, and verification of distributed avionic systems; and also the cost of certifying highly critical systems of this kind.

This work should be seen as a contribution to the broader joint effort with our colleagues at Rockwell-Collins and at UIUC to advance all aspects of the PALS architecture; in this regard, the main contribution here is to the mathematical foundations of PALS as a formal architectural pattern with strong correctness guarantees. We refer to [5] for a broader discussion of PALS and its applications, and also for a discussion of other research related to PALS. Here we give only some comments on one such related work, namely, the work by Tripakis et al. [9], because it has also a formal model with which we can make some comparisons. That work deals also, as ours, with the problem of mapping a synchronous architecture consisting of a synchronous interconnection of state machines to an asynchronous architecture. In their case, this is a loosely timed triggered architecture, where processes communicate asynchronously and have local clocks that can advance at different rates and where no clock synchronization is assumed. This mapping is achieved through an intermediate translation into a Kahn-like dataflow network. The main result in [9] is the preservation of streams and therefore the correctness of the asynchronous architecture's implementation of the original synchronous sytem. In some sense their result shows the robustness of their mapping, since correctness is achieved in spite of unpredictable communication delays and highly different clock rates in the different processes. The main differences with this work, and with the PALS idea more generally, is that, due to the quite minimal assumptions made on their asynchronous dataflow model, it does not seem possible to give hard real time bounds for the behavior of the asynchronous system realization; and it seems also problematic to deal

with the freshness of environment data coming from sensors that must be responded to within specific time bounds. Because our concern is with systems, such as avionics ones, whose distributed implementation must satisfy hard real-time constraints just as stringent as those of the original synchronous systems they implement, the model in [9], while very useful and flexible for correctness purposes, does not seem to meet the real-time needs of such systems.

References

1. A. Al-Nayeem, M. Sun, X. Qiu, L. Sha, S. P. Miller, and D. D. Cofer. A formal architecture pattern for real-time distributed systems. In *Proc. IEEE Real Time Systems Symposium*. IEEE, 2009. To appear.
2. E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Mart-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
4. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
5. S.P. Miller, D. Cofer, L. Sha, J. Meseguer, and A. Al-Nayeem. Implementing logical synchrony in integrated modular avionics. In *Proc. 28th Digital Avionics Systems Conference*. IEEE, 2009.
6. P. C. Ölveczky and J. Meseguer. Abstraction and completeness for Real-Time Maude. *Electronic Notes in Theoretical Computer Science*, 176(4):5–27, 2007.
7. P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.
8. L. Sha, A. Al-Nayeem, M. Sun, J. Meseguer, and P. C. Ölveczky. PALS: Physically asynchronous logically synchronous systems. Technical report, University of Illinois at Urbana-Champaign, 2009. Available at `http://hdl.handle.net/2142/11897`.
9. S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincentelli, P. Caspi, and M. DiNatale. Implementing synchronous models on loosely time triggered architectures. *IEEE Trans. on Computers*, 1, 2008.